C# Exercise collection 4 - Memory handling

Please note: The result of this exercise should be presented to a teacher for evaluation before it can be considered done.

Coding instructions:

All code for these exercises should be written in the accompanying projekt "SkalProj_DatastructuresAndMemory". In the project you will also find further instructions for implementation. Read this document first - then go to the project.

Exercises 1 - 4 are prioritised. exercises 5 and 6 are extras if you have additional time.

The questions in this exercise should be answered in a separate file or physical paper and presented to your teacher as you show them your solution.

Theory and background

Most of the memory management in .NET is self-handled, but as a programer it is a good idea to have some insight into how it all works under the hood. The memory is divided into a stack and a heap, but before we go deeper into their responsibilities, let's have a look at how they work.

The *stack*, as the name implies, could be visualised as a number of boxes stacked on top of eachother. We can't acces the lower boxes before we remove the ones on the top. The stack is always access from the top, and handles one box at a time.

The *heap* is very different. In the heap the the information is stored in a tree structure, and all items are accessible at all time - as long as we know which box we need.

The stack is responsible of tracking which *calls* and *methods* that currently is running. When one call is handled or a method has returned it will be tossed from the stack, and the next call is handled. This was the stack is self-managed. The heap on the other hand holds a lot of information but no knowledge of execution order, so here the garbage collection (GC) comes into play.

So the stack and the heap clearly handles different information, to learn more about these differences we first need to learn about the four types used i C#: *Value Types*, *Reference Types*, *Pointers* and *Instructions*.

Value Types are the types in *System.ValueType*. Listed below:

• *bool* • *byte* • *char* • *decimal* • *double*	• *enum* • *float* • *int* • *long* • *sbyte*	• *short* • *struct* • *uint* • *ulong* • *ushort*

Reference Types are the types that inherits from *System.Object* (or is the actual *System.Object.object*):

- *classes*
- *interfaces*
- *objects*
- *delegates*
- *string*

Next up is pointers. Pointers are fully handled by the *Common language Runtime (CLR)*, so it's not really something that we have to worry too much about. A *pointer* takes space in the memory and either holds the address for an other point in memory, or *null*.

Instructions is not a part of this exercise, but you should know that they exist.

So which type is stored where?
A reference type is always stored on the heap while a value type is store where it is declared. That might sound confusing, but this example hopefully helps:

```
public int AddTen(int InputInt)
{
    int result;
    result = InputInt + 5;
    return result;
}
```

In this example method (picture above) everything will be stored on the stack. Because the value types (int) are declared inside a method - and the method is added to the stack.

```
public class MyInt
{
    public int MyValue;
}
```

```
public MyInt AddTen(int InputInt)
{
    MyInt result = new MyInt();
    result.MyValue = InputInt + 5;
    return result;
}
```

However, in the example on the left, the *int MyValue* (which is a also a *value type*) will be stored in the *heap* since it is declared inside a *class*, which is a *reference type*.

All data in the first example will be erased from memory when the method it is done executing and isremoved from the stack.

In the second example the the object "*MyInt*" will stay in the heap even after the method is done and removed from the stack. It will stay in memory until the *Garbage Collector* takes care of it.

Questions:

1. In your words, how does the stack and the heap work in terms of structurally storing data? You are welcome to use examples or draw to explain their basic structures.
2. What are are *Value Types* and *Reference Types,* and how do they differ?
3. Following methods return different answers (see fig below). The first method returns 3, the second returns 4. Why? *MyInt* is the same class that is declared in example 2 above.

```
public int ReturnValue()
{
    int x = new int();
    x = 3;
    int y = new int();
    y = x;
    y = 4;
    return x;
}

0 references
public int ReturnValue2()
{
    MyInt x = new MyInt();
    x.MyValue = 3;
    MyInt y = new MyInt();
    y = x;
    y.MyValue = 4;
    return x.MyValue;
}
```

Data Structures and Memory effectiveness

When it comes to memory handling and writing memory efficient functions, it is important to have an understanding of different data structures and how and when they can be used. This is what you will do in this exercise, using both paper and code. Remember to comment the code.

As previously mentioned these exercises should be done in the provided shell-project and the questions should be answered on paper

Exercise 1: ExamineList()

A list is an *abstract data structure (not to be confused with an abstract class)* that can be implemented in many different ways. At this point you should be familiar with the *list class*, unlike *arrays lists* does not have a predetermined size, instead the size increases dynamically as the number of elements in the list increases. However, the *list class* does have an underlying *array* that you will be examining. To see the size of the underlying array use the *Capacity method* of the *list class*.

1. Finish the implementation of the *ExamineList-method* so that you can complete the examination.

2. When does the capacity of the list increase? (In other words the size of the underlying array)?
3. How much does the capacity increase?
4. Why does the capacity not increase at the same pace as elements are added?
5. Does the capacity decrease when elements are removed from the list?
6. When is it preferable to use an *array* you declare yourself instead of a list?

Exercise 2: ExamineQueue()

The data structure *Queue* works according to the *First In First Out* principle (FIFO). This means that the first added element will be the first removed.

1. Simulate the following queue **on paper**:
    a. ICA opens and the queue is empty
    b. Kalle stands in the queue
    c. Greta stands in the queue
    d. Kalle pays and leaves the queue
    e. Stina stands in the queue
    f. Greta pays and leaves the queue
    g. Olle stands in the queue
    h. …
2. Implement the method *TestQueue*. The method will simulate how a *queue* works by permitting the user to put elements into the queue (*enqueue*) and remove elements from the queue (*dequeue*). Use the *Queue-class* to help you implement the method, then simulate the ICA queue from the previous example in your program.

Övning 3: ExamineStack()

Stacks is quite similar to ques in structure, but with the very important difference that it uses the Fist In Last Out principle (FILO). So the first element that is added to the stack (pushed) will be the last element to leave the stack (pop).

1. Once again try to simulate the ICA-queue but this time using a stack. Why is it not a good idea to use a stack for this example?
2. Implement a ReverseText-method that takes a string input from the user. Then uses a stack to change the order of the characters and outputs them in the reverse order from how they were entered.

Övning 4: CheckParenthesis()

Use what you have learnt in previous exercises to solve the following problem:

We call a string “well formed” if all parentheses that are opened also is closed. That a parentheses is opened and closed correctly is checked by the following rules:

-),] and } is only allowed in the string after (, [and {, respectivly.
- Every parenthesis that is opened must also be closed. i.e. “(“ is followed by “)”

The string “`( [ { } ] ( { } ) )`” is considered well formed but “`( [ } )`” isn’t. The string can contain other characters as well, but they will be ignored in the check - we

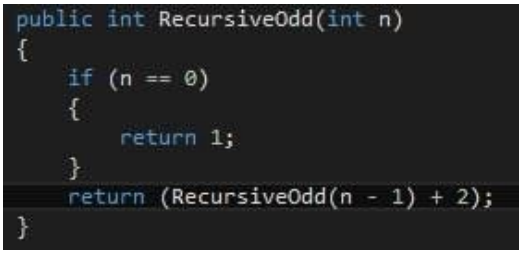
only care about the parentheses. Other examples of well formed strings are: "`List<int> CoolNumbers = new List<int>(){2, 3, 4};`" or "`( ab [ c ] )`"

1. Start on paper using sketches to solve the problem. Which data structure is best suited to help you?
2. Implement the functionality of the method CheckParentheses(). Let the user input a string and use the method to return an answer that tells the user whether the string was well formed or not.

Iteration types: Recursion and iteration

In this section we will look at the difference between recursion and iteration, and how we sometimes can get in trouble by using them unwisely.

Recursion is when a function calls itself until it reaches a base case, once the base is reached it can return all the calls until the one that initiated the recursion. The following example calculates the n:th odd number and returns the answer.



```
public int RecursiveOdd(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return (RecursiveOdd(n - 1) + 2);
}
```

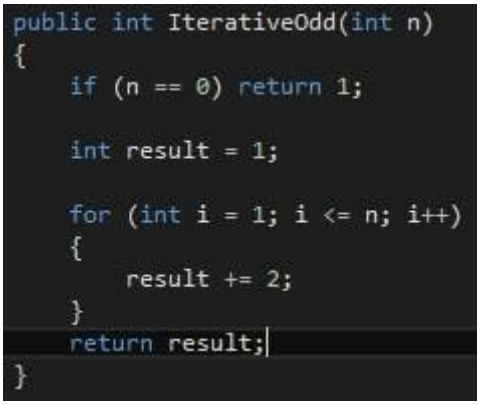
The method checks if the parameter n is zero, if so it returns the first odd number: 1. If the parameter n isn't zero it will call itself again with the parameter n-1 and adds 2 to the result.

Exercise 5: Recursion

1. Illustrate the execution of `RecursiveOdd(1)`, `RecursiveOdd(3)` and `RecursiveOdd(5)` on paper to understand the recursive loop.
2. Implement the method `RecursiveEven(int n)` the uses recursion to calculate the n:th even number.
3. Implement a recursive method to calculate the n first numbers in the fibonacci sequence: `(f(n) = f(n-1) + f(n-1))`

Exercise 6: Iteration

An iterative function repeats the same action until an end condition is met. An iterative solution the too previous example IterativeOdd could look like this:



```
public int IterativeOdd(int n)
{
    if (n == 0) return 1;

    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result += 2;
    }
    return result;
}
```

This method returns 1 if the parameter n is zero, otherwise it starts the iteration and adds 2 until the result is the n:th odd number which then is returned.

1. Illustrate the execution of `IterativeOdd(1)`, `IterativeOdd(3)` and `IterativeOdd(5)` to understand the iteration.
2. Write the method `IterativeEven(int n)` to iteratively calculate the n:th even number.
3. Implement an iterative version of your fibonacci counter.

Question:

Which of your implementations is most efficient in terms of memory - and why?